Published in Proc. SIGMETRICS 2002

Dynamic Statistical Profiling of Communication Activity in Distributed Applications

Jeffrey Vetter

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA 94551
vetter3@llnl.gov

ABSTRACT

Performance analysis of communication activity for a terascale application with traditional message tracing can be overwhelming in terms of overhead, perturbation, and storage. We propose a novel alternative that enables dynamic statistical profiling of an application's communication activity using message sampling. We have implemented an operational prototype, named PHOTON, and our evidence shows that this new approach can provide an accurate, low-overhead, tractable alternative for performance analysis of communication activity. PHOTON consists of two components: a Message Passing Interface (MPI) profiling layer that implements sampling and analysis, and a modified MPI runtime that appends a small but necessary amount of information to individual messages. More importantly, this alternative enables an assortment of runtime analysis techniques so that, in contrast to post-mortem, trace-based techniques, the raw performance data can be jettisoned immediately after analysis. Our investigation shows that message sampling can reduce overhead to imperceptible levels for many applications. Experiments on several applications demonstrate the viability of this approach. For example, with one application, our technique reduced the analysis overhead from 154% for traditional tracing to 6% for statistical profiling. We also evaluate different sampling techniques in this framework. The coverage of the sample space provided by purely random sampling is superior to counter- and timer-based sampling. Also, PHOTON's design reveals that frugal modifications to the MPI runtime system could facilitate such techniques on production computing systems, and it suggests that this sampling technique could execute continuously for long-running applications.

1 INTRODUCTION

To fully realize the potential of terascale computing, users must be able to understand the performance of their applications. Unfortunately, the scale of new systems, which will have thousands, if not millions, of processors [1], is quickly outstripping the capabilities of traditional performance analysis techniques. While traditional trace-based techniques for analyzing communication performance of distributed applications have demonstrated advantages [6, 8, 12-14, 17, 18, 20, 21, 24, 26], their operation on terascale platforms presents several challenges. In particular, these techniques require post-mortem analysis of potentially massive tracefiles, which, in turn, can lead to high instrumentation overhead and flawed performance observations.

Put simply, this paper proposes a novel alternative that addresses these challenges by enabling statistical profiling for individual messages of an application's communication activity during execution. Similar to other statistical profiling techniques [2, 3, 9], our technique strikes a balance between the comprehensive detail of tracing and the insight necessary for optimization. Our alternative enables runtime analysis of communication activity by appending a small amount of performance data to sampled messages. Then, messages can be sampled and analyzed with a variety of techniques that are easily interchanged at the Message Passing Interface (MPI) profiling layer. Also, this alternative enables an assortment of runtime analysis techniques not previously available with post-mortem techniques. In this new structure, the system can jettison raw performance data as soon as the runtime analysis is complete. Evidence from an operational prototype, built on the MPI, shows that this new technique can provide an accurate, low-overhead, tractable alternative for performance analysis.

1.1 Motivating Example

To motivate the demands of performance analysis with large-scale applications, we consider a case study of SMG2000, an MPI application with demonstrated scalability to four thousand processors. (Section 4 provides complete details of the experimental evaluation.) The goal of this example is to outline the process of traditional performance analysis, highlight its limitations, and argue for statistical profiling of communication activity via message sampling.

Trace-based performance analysis of distributed applications is very useful because it provides users with detailed chronology of their application's execution [6, 8, 13, 14, 17, 18, 20, 21, 24, 26]. As illustrated in Figure 1, the typical operation of a trace-based tool for analyzing communication operations on a distributed application is a multi-step process. To make this process more concrete, we applied a widely used MPI tracing tool to SMG2000 on 48 tasks. This application sets up and solves a linear system, a task common to scientific computing. Our tracing tool interposes instrumentation between the application and the MPI runtime using the MPI profiling layer that captures pertinent performance data as in Figure 1. As the instrumented application executes on a distributed platform, this profiling layer intercepts significant MPI subroutines, captures a timestamp and relevant subroutine parameters, records the event to a memory buffer, and, eventually, writes them to a local disk file (steps ② and ⑥). Unsurprisingly, most of this instrumentation has been optimized to use efficient buffering techniques, low overhead timers, and minimal data collection. At application termination, the tracing tool automatically merges the distributed files into one file (step ⑧), during which it sorts the events by global time, reconciles point-to-point message operations, matches collective operations, and calculates communication statistics.

Clearly, certain types of performance analysis cannot occur until this final step because some metrics, such as message latency, are not available until reconciliation of event records for point-to-point messages. Lastly, with the merged file in hand, a user can proceed with the investigation using a variety of techniques (step ⑨) including statistical analysis, pattern recognition, automated classifiers, and visualization to glean important insights into their application performance. Practically all of the popular tools in this area rely on visualization to assist users [4, 13, 23].

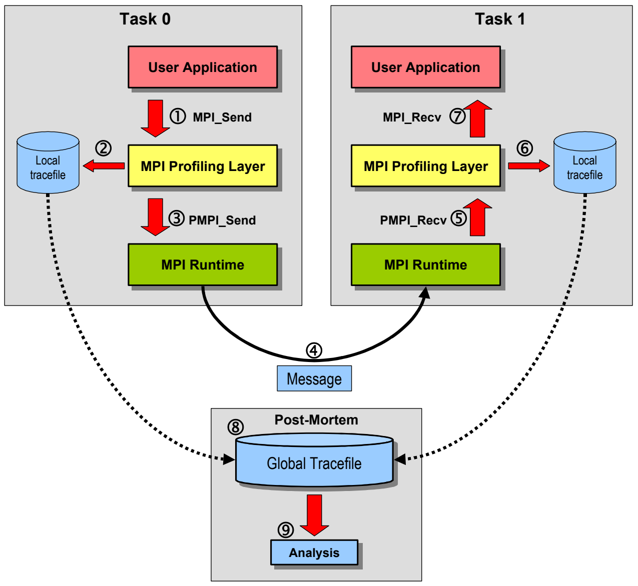

Figure 1: Traditional Performance Analysis of Communication Activity.

1.2 Observations

The first and most significant observation is that SMG2000 has an astonishing number of communication operations: it sends approximately 16,000 messages per task per solve for our example input problem. This volume of message traffic creates very large local and global tracefiles. For example, the final, merged, binary tracefile for SMG2000 on 48 tasks is 225MB. SMG2000 has scaled to over 4,000 tasks. We cannot reasonably expect to capture communication activity at ever-increasing scale using traditional tracing techniques because the amount of data generated can be intractable to store and analyze.

More problematic is that the execution time of SMG2000's solution phase increases from 26 seconds to 66 seconds – a 154% increase. This increase is due to the cumulative effect of the software instrumentation and the fact that tracing event buffers must be flushed to disk frequently.

Even when a user exercises considerable care in focusing the instrumentation on particular subroutines or on limited phases of execution, this situation can perturb the underlying application to an extent that it does not resemble the actual execution of the optimized application, especially when tracing a sequence of messaging operations. For instance, Figure 2 illustrates a sequence of communication operations that can be influenced by instrumentation perturbation. One cause of performance problems in MPI applications is the handling of unexpected incoming messages. Typically, users try to optimize their applications by posting receives before their matching sends (as for messages ① and ② in Figure 2). Tracing tools would certainly provide insight into this phenomenon; however, the intervening instrumentation may very well delay a properly posted message receive into an unexpected incoming message. In this example, the cumulative instrumentation of *a* and *b* could very well impede the MPI_Recv and make message ③ appear as an unexpected message, when in reality, without the instrumentation, it is not. Statistical profiling of messages helps to alleviate this issue in two ways. First, it reduces the instrumentation overhead because fewer messages are measured. Second, statistical profiling randomly distributes the instrumentation overhead across the entire message population, helping to avoid pathological situations like the one proposed in Figure 2.

2 ENABLING RUNTIME ANALYSIS OF COMMUNICATION ACTIVITY

As identified in Section 1.2, one major limitation of current techniques is that much of the analysis must be postponed until the distributed tracefiles can be merged and reconciled. In contrast to these trace-based techniques, we propose a new alternative that enables runtime analysis of communication activity by appending a minuscule amount of performance information to individual messages exchanged by the application. Our prototype, named PHOTON, implements these frugal modifications to enable runtime analysis and statistical sampling of messages, both of which are important elements in eliminating the burdens incumbent on trace-based techniques: overhead, perturbation, and data management.

PHOTON, our operational prototype for statistical sampling of application communication activity, focuses on the Message Passing Interface (MPI) [10, 22]. Historically, users have written scientific applications for large, distributed memory computers using explicit communication as the programming model. This trend crystallized with the creation of the MPI specification [10, 22], which simplified numerous issues for both application

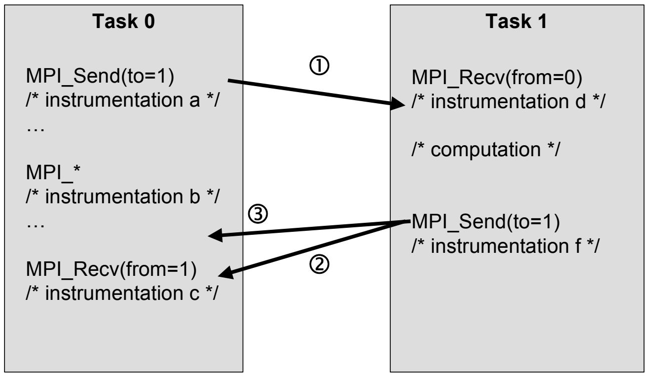

Figure 2: Perturbation example.

developers and system designers. As a result, application developers stabilized on the MPI programming model, and this has facilitated the ongoing development of a considerable number of applications based on MPI [25]. MPI provides a wide variety of communication operations including point-to-point operations, both blocking and non-blocking, and collective operations such as broadcast and global reductions. We concentrate on basic point-to-point operations—blocking send, blocking receive, non-blocking send, and non-blocking receive—because they are widely used and are the most important yet difficult components to sample.

2.1 Components

PHOTON's design has two basic components: a MPI profiling library and a modified MPI runtime library. An application must use both components to benefit from runtime analysis and message sampling. This design minimizes modifications to the underlying MPI runtime while retaining considerable flexibility at the MPI profiling layer for implementing sampling and analysis techniques. Most MPI performance analysis techniques, including traditional tracing tools, use the MPI profiling layer alone to gather performance information.

As Figure 3 illustrates, the first component of PHOTON is the modified MPI runtime library. Our implementation is a fully functional version of MPICH 1.2.2, configured to use IBM's Message Passing Library (MPL). Our modifications are minimal. We change the definition of headers for point-to-point messages and the definition of the MPI_Status record to include two new fields: a timestamp and a source code location identifier. We also modify the send and receive operations for all message protocols.

The modified send operations copy these two variables into the header of the outgoing message. For receive operations, if the incoming message is tagged, then the receive operation copies the timestamp and source code location into these two MPI_Status fields. This new definition of MPI_Status includes our two new fields, since each incoming message must set a status word.[1] The profiling layer can then easily check these two new fields in the MPI_Status structure to determine if, indeed, the send operation tagged the current message. Otherwise, it simply ignores them. If tagged, then the profiling layer can extract these two additional fields from the MPI_Status structure and use them for analysis.

Strictly speaking, our limited changes to the MPI runtime system include: (1) increased message header size by 12 bytes (where the original header size was approximately 48 bytes); (2) two writes to these fields in the send operation; (3) two reads of these fields in the receive operation; (4) one control branch each in the send and the receive operation; and (5) increased size of the MPI_Status structure. As our experimental results show in the next section, this overhead is imperceptible in the performance of our MPI implementation. Notice that these changes do not include procedure calls, data analysis, or sampling mechanisms; the performance analysis tool inserted at the MPI profiling layer provides all of these components, if desired.

The second component of our implementation exploits this additional information by using the MPI profiling layer to allow flexible analysis on unmodified application codes. Relinking via the profiling layer interposes the PHOTON profiling layer between the application and the PHOTON MPI runtime system, where it can intercept all MPI calls, exploit this new runtime performance information, and analyze data as necessary.

[1] The MPI specification allows users to set MPI_STATUS_IGNORE to bypass the setup of the MPI_Status structure. PHOTON creates a temporary MPI_Status structure and passes this structure to the underlying system.

2.2 Operation

Given these two components, PHOTON works as illustrated in Figure 3. A message is sent from Task 0 to Task 1 using MPI's blocking communication routines. At step ①, the application prepares the message and calls the MPI_Send routine. The MPI_Send routine is intercepted by the MPI profiling layer. At point ②, PHOTON decides whether to sample this particular message. Assume that it decides to sample the current message. PHOTON then records the start timestamp of the send operation and an identifier describing this operation's location in the source code. Usually, this location is the return address of this MPI subroutine; however, it can be a more elaborate hash function that encodes a stack traceback, message parameters, etc. PHOTON passes these two additional pieces of information on to the MPI runtime and calls the name-shifted profiling layer routine, PMPI_Send. As the MPI runtime begins at step ③, it loads this additional information into the message header that it prepares. It then dispatches this message to the underlying message libraries. As the tagged message flows across the network at step ④, it carries these two

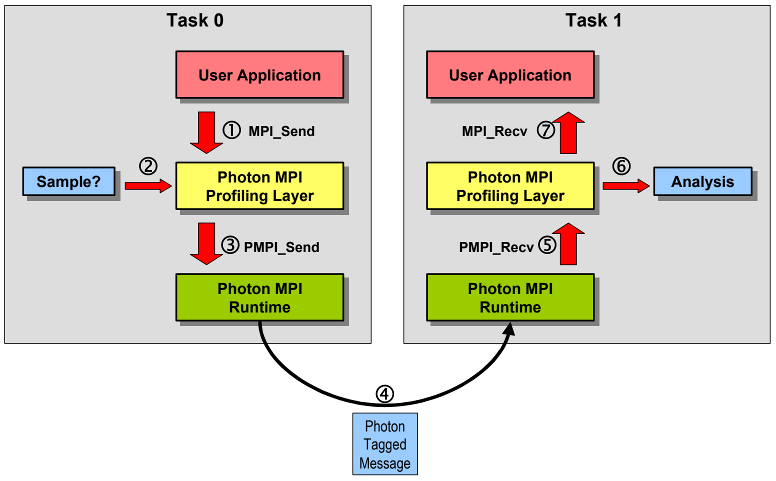

Figure 3: PHOTON Design Overview.

additional pieces of information with it in its message header. This operation is similar for all MPI protocols.

Meanwhile, Task 1 has issued a blocking MPI_Recv for this message, not knowing if the incoming message is a tagged message or not. As the user application calls MPI_Recv at step ⑦, it must post the blocking PMPI_Recv at step ⑤ without knowledge of the sampling decision.[2] Only when the message is received can Task 1 actually make a decision about how to handle this message. It is important to note that this problem is impossible to solve for all MPI protocols within the MPI profiling layer alone, and it necessitated our modifications to the MPI runtime.

Now, Task 1 receives the message at step ④ and it recognizes that this message has been tagged, so the MPI runtime copies this information directly into the two additional fields in our modified MPI_Status structure. When the MPI runtime completes the receive of this tagged message, it returns from the PMPI_Recv into the PHOTON profiling layer. At this point, PHOTON can make any number of decisions about how to analyze the tagged message. At step ⑥, PHOTON can record statistics and

[2] Arguably, MPI_PROBE/IPROBE could provide this foresight for blocking receives; however, this strategy fails for non-blocking operations.

discard the data, write it to a trace file, or simply ignore it. When PHOTON has completed its analysis at step ⑥, it returns to the user application via the MPI_Recv call at step ⑦. The user application can then process the message as it normally would without regard to the fact that the message was sampled by the underlying performance analysis system.

2.3 Key Design Implications

This design alternative has a number of important implications. First, message processing is undisturbed. Our technique does not require additional messages, additional copying of message buffers, or excessive quantities of extra buffer space. Therefore, the MPI behaviors of this alternative should closely resemble the behaviors of the original MPI application. Although we considered several options to modifying the underlying MPI implementation, none of these strategies provided necessary functionality for all MPI operations and respected all of the requirements demanded by the MPI specification, such as its message-ordering requirement.

More specifically, three alternatives come to mind. In the first alternative, one could simply send an extra message following each sampled message. This alternative has two drawbacks: it introduces additional messages into the system, and the receiving task has no *a priori* knowledge of when to receive an extra message. This strategy could also introduce race conditions into the application. In the second alternative, the system could exchange performance data during collective operations; however, the overhead could be noticeable and reconciling sends with receives would still demand comprehensive knowledge of all point-to-point operations. The final alternative is that a technique could use MPI derived types to piggyback additional data onto messages. On the face of it, this alternative is appealing. The MPI specification allows nested data types, so the profiling layer could simply repackage a message with additional performance data, and send it to the receiver where it is unpackaged with this performance data. Unfortunately, this alternative has several problems. First, MPI-derived types can perform poorly because of additional memory copying and buffering of data, which might drastically alter the performance characteristics of the application [11]. Second, the receiving task cannot determine which messages are tagged messages. MPI message envelopes specify message source, tag, and communicator, but not data type. This limited information prohibits the receiver from determining whether an incoming message is tagged by using the envelope information. Therefore, the receive operation has no idea of how to prepare the receive buffer. This strategy is also plagued by the possibility of several types of race conditions, if message tags are used to discriminate sampled messages.

The second important implication is that decisions regarding sampling and analysis techniques remain at the MPI profiling layer. Thus, these techniques can be easily interchanged without altering the MPI runtime. Better still, by relegating all of these decisions to the profiling layer, the performance of the modified MPI runtime can remain practically unchanged from the original.

The third and final implication is that this design is applicable to all types of point-to-point communication regardless of MPI subroutine or message protocol. Take, for example, non-blocking communication. PHOTON simply loads the message header during the initiation of the non-blocking MPI_Isend. The underlying message library transfers the message normally. As the message is received with an MPI_Irecv/MPI_Wait pair, the extra performance information from the message header is transferred directly into the MPI_Status structure, which is provided to MPI_Wait as a parameter. This line of reasoning holds true for other completion operations including MPI_Test and MPI_Waitsome because they return an MPI_Status structures.

3 STATISTICAL MESSAGE SAMPLING

Although sampling is popular in many areas of performance analysis, such as procedure profiling [3, 9] and instruction analysis using hardware counters [2], it has not been applied to communication activity because of the limitations listed in Section 1. With the novel design modifications proposed in Section 0, we can now reliably and accurately access performance information at runtime, so that communication performance analysis can capitalize on the benefits of statistical sampling. To our knowledge, this technique is novel and it represents a significant shift in current technology for performance analysis of terascale, distributed applications. Sampling has also been applied at low levels of communication activity [7], but this research focused on understanding wide-area networks rather than on optimizing the applications that use those networks.

3.1 Sampling Strategies

As Figure 3 illustrates in step ②, PHOTON can use most any technique to decide which messages to sample from the entire message population of the application. During the send initiation operation for each message, PHOTON decides whether to sample a message. Naturally, three different techniques can drive our approach: purely random sampling, counter-based sampling, and timer-based sampling. Accordingly, the application must execute for a reasonably long time and it must send some minimum number of messages for the sampling to be accurate. These methods sample from the entire message population; however, they can also be adapted to sample subsets of the population. For instance, we could sample only large messages, only messages sent from a certain callsite, or only messages sent during a certain phase of the application execution.

Random sampling: Our first sampling method is purely random sampling. On every send operation, PHOTON draws a number from a uniform distribution in (0,1] and then checks that number against a user-defined threshold (T) to determine if the current message should be sampled. This strategy is simple and it allows a user to easily control the number of samples by changing the threshold.

Counter-based sampling: In PHOTON, for counter-based sampling, a single counter in each task is incremented for every send operation. When the counter exceeds a threshold, one message is sampled, the counter is reset, and a new target threshold is calculated. The user can select the period (P) and variance (V) of the counter. Counter-based sampling benefits from its simplicity and low overhead; however, estimating the appropriate settings for P and V can be difficult because it depends entirely on the frequency of application communication.

Timer-based sampling: Similarly, for timer-based sampling a message is sampled after a period of time has expired since the last send operation. The user specifies a period of time (P) and a variance (V). Note that our technique does not use expensive interrupts to execute this sampling technique. Rather, as the application calls MPI's send routines, PHOTON polls the local time and then decides if the specified threshold has been met. Although the cost of sampling the timer can be expensive relative to using counters, timer parameters are much easier to estimate and, in any

Configuration	Moniker	PHOTON MPI Runtime	PHOTON profiling layer	Sampling method	Analysis Method	Description
Original MPI	ORIG	No	No	-	-	
MPI Modified to include PHOTON information	NOPROF	Yes	No	-	-	MPI Runtime w/ PHOTON modifications. Lacks PHOTON profiling layer that includes sampling and analysis methods.
Sampling enabled at various random thresholds	P-X	Yes	Yes	Random X = threshold	-	Modified MPI with sampling infrastructure installed where the threshold for random sampling is X. No analysis is performed on the sampled message.
Sampling enabled at various counter rates	C-X	Yes	Yes	Counter X = period	-	Modified MPI with sampling infrastructure installed where the counter-sampling period is X. No analysis is performed on the sampled message.
Sampling enabled at various timer periods	T-X	Yes	Yes	Timer X = period	-	Modified MPI with sampling infrastructure installed where the timer-sampling period is X. No analysis is performed on the sampled message.
Sampling w/ various analysis methods	A-X-B	Yes	Yes	A = sampling method X = threshold	B is WRITE, STAT, or FREQ	Modified MPI with sampling infrastructure installed where the sampling method is A with threshold X and the stated analysis B is performed on each sampled message.

Table 1: Experiment Configuration Overview.

case, PHOTON must capture this timestamp for the outgoing sampled message.

3.2 Analysis Methods

Now, with a considerable amount of performance information available at runtime, PHOTON can elect to perform analysis at runtime and jettison raw performance data immediately. In the context of terascale computing, this capability is vital because it eliminates the need for capturing massive tracefiles and harvesting important performance data from those files. Certain performance problems may still require tracing; however, this runtime analysis can quickly identify a subset of operations for more detailed investigation.

For performance analysis, our experience indicates that it is important that users are able to map performance data back to source code; we record and tabulate all messages by source task, destination task, callsite location in source and destination, and message size. We capture message latency as our primary performance metric, where we define latency as the time from the start of the send operation until the end of the matching receive operation. Although this definition is somewhat different than architectural definitions, this interpretation is easy for a user to reason about, and it maps directly to the user's source code. Most trace-based tools use similar definitions.

We introduce two lightweight techniques to analyze this data at runtime, and we include an expensive technique of writing the event to a local file as a reference measurement.

Statistical summary (STAT): One traditional option for profiling is a statistical summary of the messages. For our statistical summary, PHOTON captures a maximum, a minimum, a count, and a cumulative total of the message latency. With a long application execution, this statistical information would supply a reasonably accurate picture of communication activity. It also has a very small analysis overhead that includes locating an entry in a data structure and updating several fields in that entry. This summary, when combined with the message size and topology information, can identify performance problems of specific message operations.

Frequency distributions (FREQ): Another valuable technique for analyzing large masses of raw data is a frequency distribution. Using a frequency distribution, an analyst can project any number of raw data samples into classes, which are easy to represent, understand, and store. These distributions provide more information than the statistical summary presented above; however, the overhead for updating the entry and the memory requirements is increased slightly. As before, we categorize the data by sender, receiver, callsite locations for both the sender and receiver, and message size. Our implementation creates an array of bins for message latency delimited by a common log scale. Each bin counter is incremented when a message's latency falls within that bin's bounds.

Write to File (WRITE): Although our main focus for PHOTON is runtime analysis of performance data, we can simply use the analysis step to write the values to a memory buffer or a file. In contrast to traditional tracing, however, this file would not represent a complete chronology of the application's communication behavior, but rather a small, sampled portion of the overall communication.

4 EVALUATION

Our evaluation focuses on the hypothesis that PHOTON can offer a feasible alternative to trace-based analysis of message-passing applications on terascale platforms. In this regard, we evaluate PHOTON along several dimensions: overhead and perturbation, sampling and analysis methods, and improvements in data management. Then, we apply PHOTON to several applications in a realistic situation. Table 1 provides an overview of our experiment configurations.

4.1 Platform

We ran our tests on an IBM SP system. This machine is composed of sixteen 222 MHz IBM Power3 eight-way SMP nodes, totaling 128 CPUs. Each processor has three integer units, two floating-point units, and two load/store units. Its 64 KB L1 cache is 128-way associative with 32 byte cache lines, and L1 uses a round-robin replacement scheme. The L2 cache is 8 MB in size, which is four-way set associative with its own private cache bus. At the time of our tests, the batch partition had 15 nodes and the operating system was AIX 4.3.3. Each SMP node contains 4 GB main memory for a total of 64 GB system memory. A Colony switch—a proprietary IBM interconnect—connects the nodes. We compiled the various tests with the IBM XL compilers. Our test jobs ran on dedicated nodes, although other jobs were concurrently using the network. We built our PHOTON prototype with the publicly available version of MPI, namely MPICH 1.2.2, which we configured to use IBM's Message Passing Layer (MPL) as the communication substrate.

4.2 Overhead

Our proposals hinge on the ability of the MPI runtime system to carry a small amount of additional information with each message. To assess the penalty for this increase in the message header size, we compare three versions of MPI as described by Table 1: (1) a version of the original MPI (ORIG); (2) a version of the modified MPI with the minimal internal changes, but without the PHOTON profiling layer installed (NOPROF); and (3) a version with both the modified MPI and the PHOTON profiling layer configured to sample no messages (P-0-OFF). For these experiments, we did not perform analysis on the resulting sampled message; we consider analysis methods in the next section. These experiments measure message latency and bandwidth for each configuration using the Pallas PMB-MPI1 PingPong benchmark between two SMP nodes. This benchmark measures average latency and bandwidth for multiple send–receive pairs, and it implements a "warm up" period prior to each test.

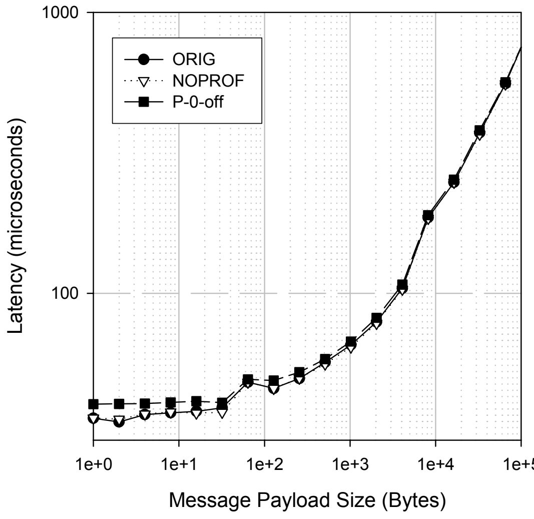

Figure 4: PHOTON Overhead.

As Figure 4 illustrates, the modified version of MPI (NOPROF) performs identically to the original version (ORIG). The increase in latency was less than one microsecond for all message sizes. When we included the PHOTON profiling library with sampling disabled (P-0-OFF), the overhead increased by three microseconds at a payload size of 4 bytes, though it is only noticeable for messages that have fewer than 10,000 bytes. As message size increases, this fixed-cost overhead disappears into the greater cost of the communication operation, as expected. Although we do expect the overhead for PHOTON to be very small for most MPI implementations, the impact of these changes will be proportionally higher on optimized messaging layers, such as those that use shared memory for intra-node transfers on an SMP. Our experiments showed no measurable change in the communication bandwidth across these configurations.

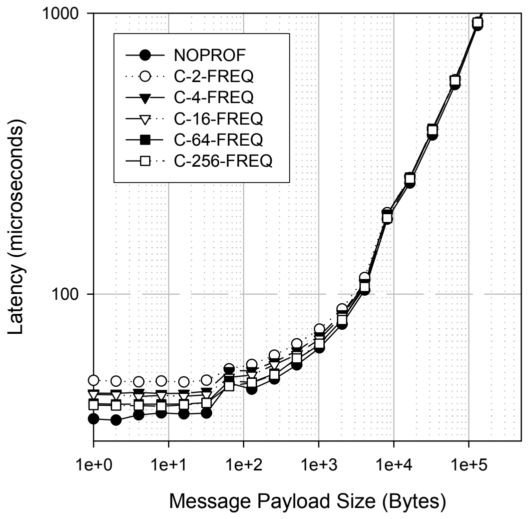

Figure 5: PHOTON Overhead for Counter Sampling.

This result is pivotal because it argues that these new features could be included in many MPI implementations without mandatory performance degradation. The optional profiling library does inflict a small overhead, but it is only noticeable only for small messages. This result also supports our decision to separate the sampling and analysis implementation from our changes to the internal MPI runtime for runtime analysis.

4.3 Sampling Methods

The various sampling methods introduced in Section 3.1 have different performance penalties and results with respect to how they sample the message population of the application. All three options provide a convenient means to control the number of messages sampled, and we use that feature to evaluate each method's overhead and sample space. We delay discussion of sample space until Section 4.5 where we evaluate it on real applications. As shown in Table 1, we vary the threshold (X) for random sampling (P-X), the period (X) for counter-based sampling (C-X), and the period (X) for timer-based sampling (T-X). Variance for counter- and timer-based sampling was 10% of the respective period. Each sampled message is analyzed with the frequency distribution technique.

First, for counter sampling, we range the period from 2 to 256. Figure 5 illustrates the overhead's explicit variation with the counter period. For a payload of 4 bytes, the latency ranges from 48.7 microseconds for C-2 down to 40.1 microseconds for C-256. For reference, we include the lower bound latency of 37.2 microseconds at 4 bytes for the NOPROF configuration. For applications that send messages with payloads of more than 20,000 bytes, the cost is imperceptible relative to the overall messaging cost.

Next, Figure 6 shows that the overhead for timer-based sampling is less controllable, but it remains low. A 4-byte message suffers an increase in latency from 37.2 microseconds for NOPROF to 45.5 microseconds for T-0.001-FREQ, where messages are sampled at a period of 1 millisecond. Increasing the sampling period to 300 milliseconds lowers the latency to 41.8 microseconds.

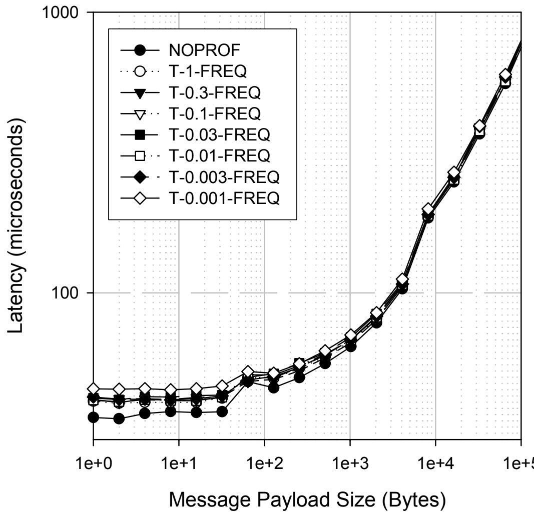

Figure 6: PHOTON Overhead for Timer Sampling.

Last, for random sampling, we range the threshold from 0.01 (1%) up to 0.9 (90%). Figure 7 illustrates the variation in response to this range. At 1% for a 4-byte message, the latency increases to 43.3 microseconds from 37.2 microseconds for NOPROF. As the threshold increases up to 90%, the latency also increases to 57.0 microseconds.

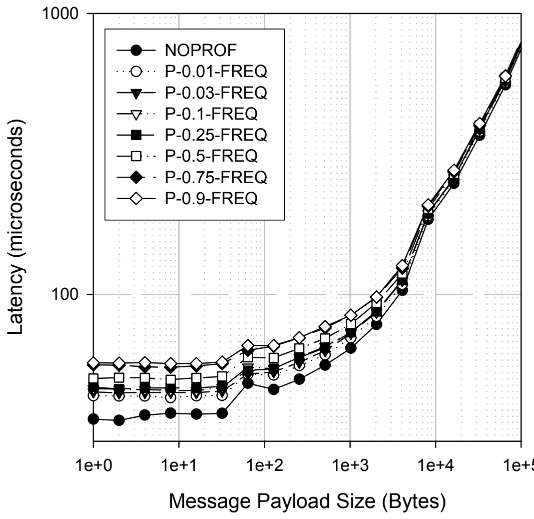

Figure 7: PHOTON Overhead for Random Sampling.

This evidence reveals that all three sampling strategies have a demonstrated controllability of overhead with respect to message latency, and it shows that each strategy can have limited impact on the target.

4.4 Analysis Methods

Runtime analysis methods are valuable because after analysis, the raw data can be purged immediately, eliminating many of the problems with traditional techniques. However, the analysis methods must balance conflicting goals of minimizing computation and storage against the goal of providing enough detailed information for a user to make a proper diagnosis. For this reason, we evaluate two lightweight statistical analysis techniques and a traditional technique of writing the data to a local file. Viewed in this light, it is important to remember that the cost of the analysis technique can be balanced against the sampling rates evaluated earlier. If a particular analysis method has relatively high cost, then the sampling rate can be lowered to compensate, minimizing the overall impact on the application execution.

As Figure 8 illustrates, the overhead of analysis methods varies considerably; however, our ability to change the sampling rates provides considerable flexibility in regulating this analysis cost. In Figure 8, we see that the most expensive technique is writing a record to a file (P-0.5-WRITE) at 59.9 microseconds, and as we would expect, that the lowest overhead occurs when analysis is disabled (P-0.1-OFF). This cost drops as we decrease the random sampling threshold to 10% (P-0.1) from 50% (P-0.5) for each analysis technique. Although both the statistical summary method and the frequency distribution method yield similar levels of overhead, they perform much better that the write technique. For applications that execute for long periods of time and send a large volume of messages, a user could lower the sampling rate to a very small threshold (e.g., 0.1%).

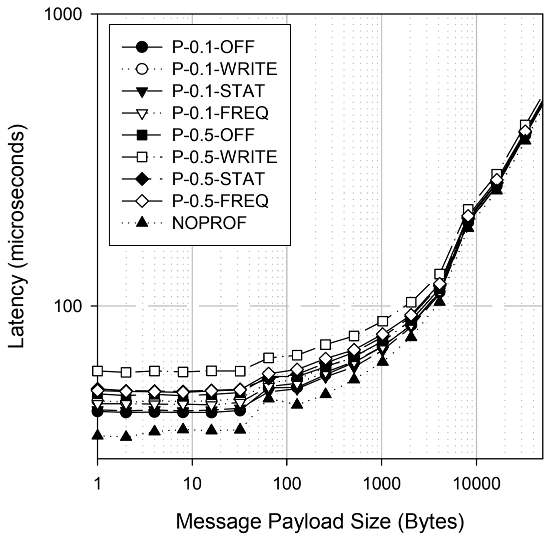

Figure 8: PHOTON Overhead for Analysis Methods.

4.5 Applications

To verify that our new technique works with real applications, we tested three applications: sPPM [19], Sweep3D [16], and SMG2000 [5]. All of these benchmarks have scaled to thousands of tasks, use MPI, are publicly available, and use point-to-point primitives for the majority of their communication. In these examples, we scaled problem sizes with the number of tasks, keeping the amount of work per task constant.

sPPM: sPPM [19] solves a 3-D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the Piecewise Parabolic Method. The algorithm makes use of a split scheme of X, Y, and Z Lagrangian and remap steps, which are computed as three separate sweeps through the mesh per timestep.

Message passing provides updates to ghost cells from neighboring domains three times per timestep.

Sweep3D: Sweep3D [15, 16] is a solver for the 3-D, time-independent, particle transport equation on an orthogonal mesh, and it uses a multidimensional wavefront algorithm for "discrete ordinates" deterministic particle transport simulation. Sweep3D benefits from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three-dimensional space is decomposed onto a two-dimensional orthogonal mesh, where each processor is assigned one columnar domain. Sweep3D exchanges messages between processors as wavefronts propagate diagonally across this 3-D space in eight directions.

SMG2000: SMG2000 [5] is a parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation $\nabla \cdot (D\nabla u) + \sigma u = f$ on logically rectangular grids. The code solves both 2-D and 3-D problems with discretization stencils of up to 9-points in 2-D and up to 27-points in 3-D. Because SMG2000 is an implicit solver, its message characteristics are much different than both sPPM and Sweep3D. SMG2000 sends considerably more messages and the messages are orders of magnitude smaller, usually less than 2,000 bytes. The message distribution is also much different. sPPM and Sweep3D have regimented 3-D and 2-D nearest neighbor communication patterns, respectively. On average, every SMG2000 task communicates with one half of all other tasks.

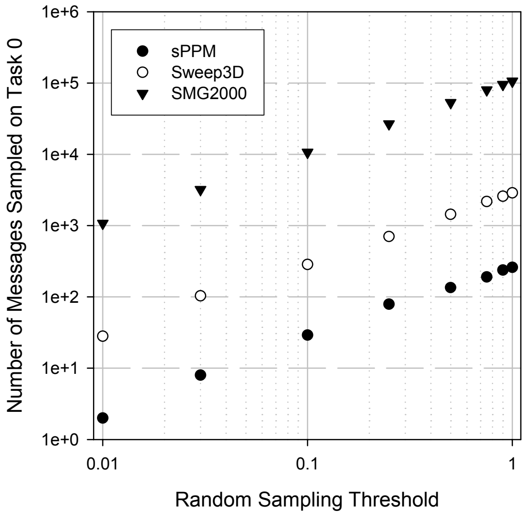

Figure 9: Number of Messages Sampled on Task 0.

Figure 9 shows the number of messages actually sampled by PHOTON on our three applications, where a threshold of 1.0 shows the total number of messages sent from Task 0. Here, we can also see the differences in the number of messages sent by every application. SMG2000 sends about 105,000 messages whereas sPPM sends only 260. Encouragingly, Figure 9 illustrates that PHOTON can easily change the number of messages sampled during an application experiment.

Sweep3D and sPPM send relatively few messages; our experiments confirmed that all sampling rates less than 50% did not have a measurable effect on the application runtime. When the sampling threshold increased above 50%, the execution time of both applications increased consistently but never more than 1%. SMG2000, on the other hand, sends significantly more messages; Figure 10 shows the impact of the PHOTON sampling threshold on the SMG2000 runtime, normalized to the NOPROF configuration. We see that initially, at a 1% sampling threshold, SMG2000 experiences a slowdown of 6.1%. As the sampling threshold increases to 90%, the slowdown climbs to 11%.

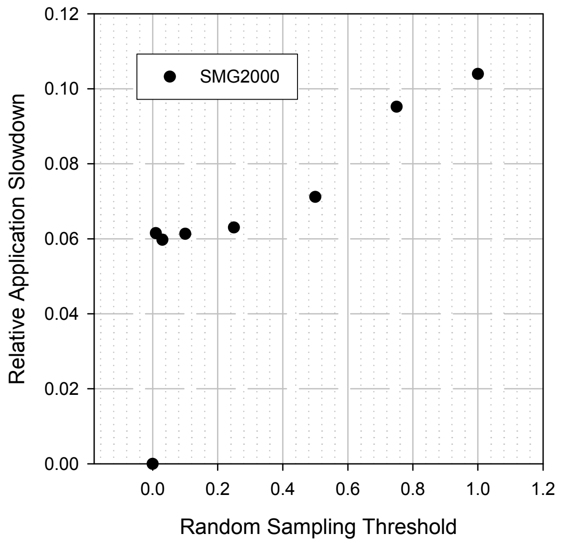

Figure 10: Impact of sampling rate on SMG2000 runtime.

Although SMG2000 does incur perturbation on the application runtime, its impact is much less than the traditional tracing approach that had a 154% slowdown in execution time. Much more importantly, because PHOTON gathers its information with sampling, it helps to avoid pathological measurement problems like those mentioned in Section 1.2. Still, we believe that more aggressive optimization of the PHOTON profiling layer, which includes the sampling and analysis logic, would reduce this overhead. For instance, the current implementation calls the system-wide global clock to timestamp outgoing messages, and consequently, we sample it at the beginning of every send operation. An implementation with lower overhead might use a hybrid timer scheme where the expensive system-wide global clock is accessed periodically and the local, economical clock is used between these global clock updates. Also, our current strategy uses the standard UNIX libraries for its floating-point random number generators; we could possibly reduce their expense by using different operations, and by changing the implementation of these generators.

Another characteristic of different sampling methods and thresholds is the resulting sample space. Figure 11 shows the relative number of distinct messages sampled as a portion of the entire sample spaces for SMG2000 and sPPM. Here, we define a distinct message as at least one sample of a message identifier, where a *message identifier* is a unique identifier hashed from the tuple of source rank, destination rank, source code location, destination code location, message size. Clearly, as the sampling threshold increases, so does coverage of the population. However, PHOTON's sampling rate is controlled for each task, not the application as a whole. When samples are aggregated across all tasks, with a 25% threshold, we get over 80% coverage for each of these experiments. We expect this coverage to increase for long-running executions of any application. Admittedly, our strategy suffers from issues with sampling: messages that are sent infrequently, or only once, might never be sampled.

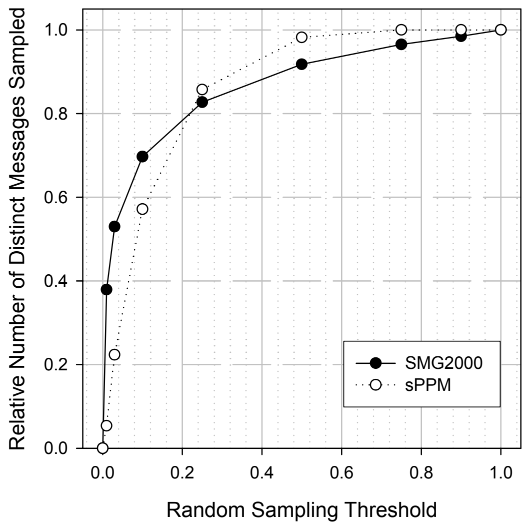

Figure 11: Sample Space of Message Population.

Figure 12 shows a subset of the performance data from our experiments on SMG2000. *Latency distribution* is a frequency distribution of the message latency where bin bounds are delimited by a common log scale. In this figure, we have displayed 36 of over 11,000 message identifiers. The distributions at *A* and *B* show message latency falling primarily into one bin, while the message latency for *C* has a wider distribution. A user could easily harvest this distribution information, combine it with knowledge of the topology, and highlight poor communication activity (including the locations of the respective source code) as well as the consistency of specific measurements.

4.6 Observations

Our experimental evaluation substantiated many of our ideas and it also revealed several issues. Among the most important observations exposed by our experiments were that by using a variety of sampling and analysis techniques, we could control the performance overhead on message latency, bandwidth, and application runtime. Furthermore, the design of our PHOTON prototype underscores that a clean separation of functionality between the MPI runtime and the MPI profiling layer can preserve performance and flexibility.

Reduced overhead. Our evaluation revealed that we could dramatically control the overhead of PHOTON while maintaining important information like message latency with statistical profiling. Our experiments on real applications and on benchmarks demonstrated that this approach radically diminishes the impact of instrumentation on the application. For SMG2000, we reduced the impact on solver runtime from a 154% slowdown for tracing to 6% for message sampling. Consequently, these variable sampling rates can be balanced against the cost of the analysis technique.

Improved accuracy. As a result of the reduced overhead, this approach improves the accuracy of the performance analysis. By reducing instrumentation impact and by randomly sampling individual messages, PHOTON provides a more accurate analysis of communication activity and it can paint a vastly different picture than traditional tracing communication. Preliminary comparisons of tracefile data to message sampling data show that different analysis conclusions are possible, and our future work includes quantifying these differences.

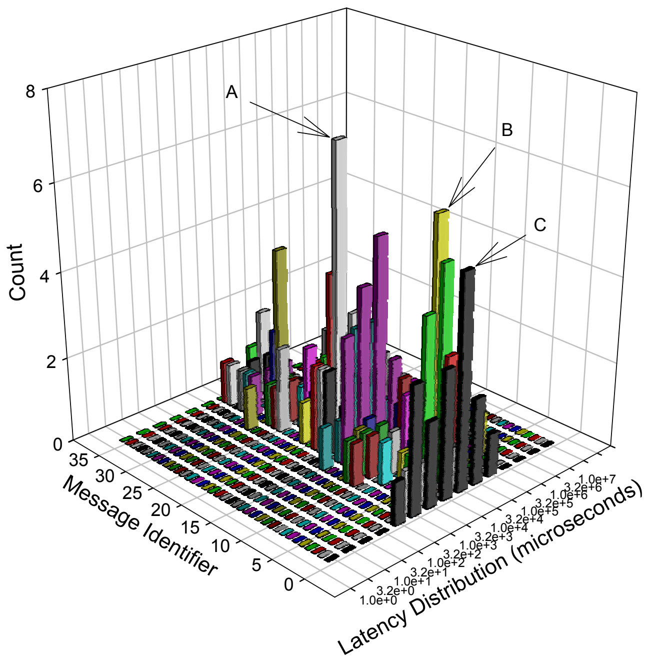

Figure 12: Example Frequency Distribution.

Reduced data volume. Statistical message sampling, when combined with runtime data analysis, eases the data management burden imposed by traditional techniques; our experiments on several applications demonstrate that significant, useful performance information can be processed dynamically. Because we can calculate message latency at runtime, the analysis can proceed immediately, discarding any raw performance data as necessary. Our lightweight analysis examples—statistical summary and frequency distribution—show that PHOTON can collect important information from long-running, terascale applications.

Frugal modifications to MPI runtime have negligible performance implications. The design of our PHOTON prototype emphasizes that a clean separation of functionality between the MPI runtime and the MPI profiling layer can preserve performance and flexibility. Changes to the MPI runtime system were frugal and our evidence indicates that these changes have an almost imperceptible performance penalty. Control of the sampling and analysis techniques remains in the profiling layer, so that users can simply interpose various techniques into their applications to do performance analysis. Indeed, we believe this result argues for PHOTON features in production MPI implementations on terascale systems because it will be necessary for any performance analysis of communication activity. What is more, with its low overhead and amortized perturbation, this technique is also appealing to consider using it continuously for large-scale, long-running applications.

Sampling methods. We found that our timer-based sampling method can lead to pathological situations where only certain messages are sampled. For example, it is common in scientific simulations to exchange data with neighbors during one phase of a timestep and compute during the other phase. We found that many of our experiments with timer-based sampling sampled only the first communication operation immediately after the compute phase, and then missed several of the consecutive send operations that immediately followed it. The other two sampling strategies did not suffer from this deficiency because they operate independent of time. More importantly, our random sampling strategy allows a user to specify a uncomplicated percentage of messages to sample.

5 CONCLUSIONS

As parallel computing systems continue to scale to massive numbers of processors, performance analysis techniques must allow users to understand their application's behaviors. We have proposed and evaluated a novel alternative to trace-based, post-mortem performance analysis of communication activity for these applications: statistical profiling of communication activity using runtime message sampling. Our operational prototype, PHOTON, demonstrated several significant advantages over the traditional trace-based approaches including dramatically lower overhead and perturbation, even for applications that communicate frequently, such as SMG2000. Our implementation strategy required minor modifications to the MPI runtime to enable runtime analysis of performance data, and it allowed flexible development of different sampling and analysis techniques at the MPI profiling layer.

ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. This paper is available as Technical Report UCRL-JC-145800.

REFERENCES

[1] G.S. Almasi, C. Cascaval *et al.*, "Demonstrating the scalability of a molecular dynamics application on a Petaflop computer," Proc. Int'l Conf. Supercomputing, 2001, pp. 393-406.

[2] J.M. Anderson, L.M. Berc et al., "Continuous profiling: where have all the cycles gone?," ACM Trans. Computer Systems, 15(4):357-90, 1997.

[3] T.E. Anderson and E.D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance," Proc. 1990 SIGMETRICS Conf. Measurement and Modeling Computer Systems, 1990, pp. 115-25.

[4] R. Bosch, C. Stolte *et al.*, "Rivet: a flexible environment for computer systems visualization," *Computer Graphics*, 34(1):68-73, 2000.

[5] P.N. Brown, R.D. Falgout, and J.E. Jones, "Semicoarsening multigrid on distributed memory machines," *SIAM Journal on Scientific Computing*, 21(5):1823-34, 2000.

[6] J. Caubet, J. Gimenez *et al.*, "A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications," Proc. Workshop on OpenMP Applications and Tools (WOMPAT), 2001.

[7] K.C. Claffy, G.C. Polyzos, and H.-W. Braun, "Application of sampling methodologies to network traffic characterization," Proc. SIGCOMM: Communications architectures, protocols and applications, 1993, pp. 194-203.

[8] G.A. Geist, M.T. Heath *et al.*, "A Users' Guide to PICL - A Portable Instrumented Communication Library," Oak Ridge National Laboratory, P.O.Box 2009, Bldg. 9207-A, Oak Ridge, TN 37831-8083 1991.

[9] S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: A Call Graph Execution Profiler," *SIGPLAN Notices (SIGPLAN '82 Symp. Compiler Construction)*, 17(6):120-6, 1982.

[10] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, 2nd ed. Cambridge, MA: MIT Press, 1999.

[11] W.D. Gropp, E. Lusk, and D. Swider, "Improving the Performance of MPI Derived Datatypes," Proc. MPI Developers and Users Conference (MPIDC), 1999.

[12] W. Gu, G. Eisenhauer *et al.*, "Falcon: On-line Monitoring and Steering of Parallel Programs," *Concurrency: Practice and Experience*, 10(9):699-736, 1998.

[13] M.T. Heath, A.D. Malony, and D.T. Rover, "Parallel performance visualization: from practice to theory," *IEEE Parallel & Distributed Technology: Systems & Applications*, 3(4):44-60, 1995.

[14] J. Hoeflinger, B. Kuhn *et al.*, "An Integrated Performance Visualizer for OpenMP/MPI Programs," Proc. Workshop on OpenMP Applications and Tools (WOMPAT), 2001.

[15] A. Hoisie, O. Lubeck *et al.*, "A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs," Proc. ICPP 2000, 2000.

[16] K.R. Koch, R.S. Baker, and R.E. Alcouffe, "Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor," *Trans. Amer. Nuc. Soc.*, 65(198), 1992.

[17] J. Labarta, S. Girona *et al.*, "DiP: A Parallel Program Development Environment," CEPBA, Barcelona, Spain 1996.

[18] A.D. Malony and D.A. Reed, "Visualizing Parallel Computer System Performance," in *Parallel Computer Systems: Performance Instrumentation and Visualization*, M.S. Bucher, Ed. New York: ACM, 1990.

[19] A.A. Mirin, R.H. Cohen *et al.*, "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System," Proc. SC99: High Performance Networking and Computing Conf. (electronic publication), 1999.

[20] D.A. Reed, P.C. Roth *et al.*, "Scalable performance analysis: the Pablo performance analysis environment," Proc. Scalable Parallel Libraries Conf., 1994, pp. 104-13.

[21] S. Shende, A.D. Malony *et al.*, "Portable profiling and tracing for parallel, scientific applications using C++," Proc. SIGMETRICS Symp. Parallel and Distributed Tools (SPDT), 1998, pp. 134-45.

[22] M. Snir, S. Otto *et al.*, Eds., *MPI--the complete reference*, 2nd ed. Cambridge, MA: MIT Press, 1998.

[23] J. Stasko, J. Domingue *et al.*, Eds., *Software Visualization: Programming as a Multimedia Experience,*. Cambridge, MA: MIT Press, 1998.

[24] J.S. Vetter, “Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies,” Proc. ACM Int'l Conf. Supercomputing (ICS), 2000, pp. 245 - 54.

[25] J.S. Vetter and F. Mueller, “Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures,” Proc. International Parallel and Distributed Processing Symposium (IPDPS), 2002.

[26] C.E. Wu, A. Bolmarcich *et al.*, “From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems,” Proc. SC2000: High Performance Networking and Computing, 2000.